Talking About My Generation: Generating Targeted Cross-site Scripting Exploits using Dynamic Data-Flow Analysis

EuroSec

26 April 2021

SAP Security Research

About us

Souphiane Bensalim
SAP Security Research
souphianebensalim@gmail.com

Thomas Barber
SAP Security Research
thomas.barber@sap.com

David Klein
Technische Universität Braunschweig
david.klein@tu-braunschweig.de

Martin Johns
Technische Universität Braunschweig
m.johns@tu-braunschweig.de

Agenda

- Motivation
- Tainting in the Client Side
- Automated Generation of Cross-site Scripting Exploits
- Large Scale Crawling and Results
- Conclusion

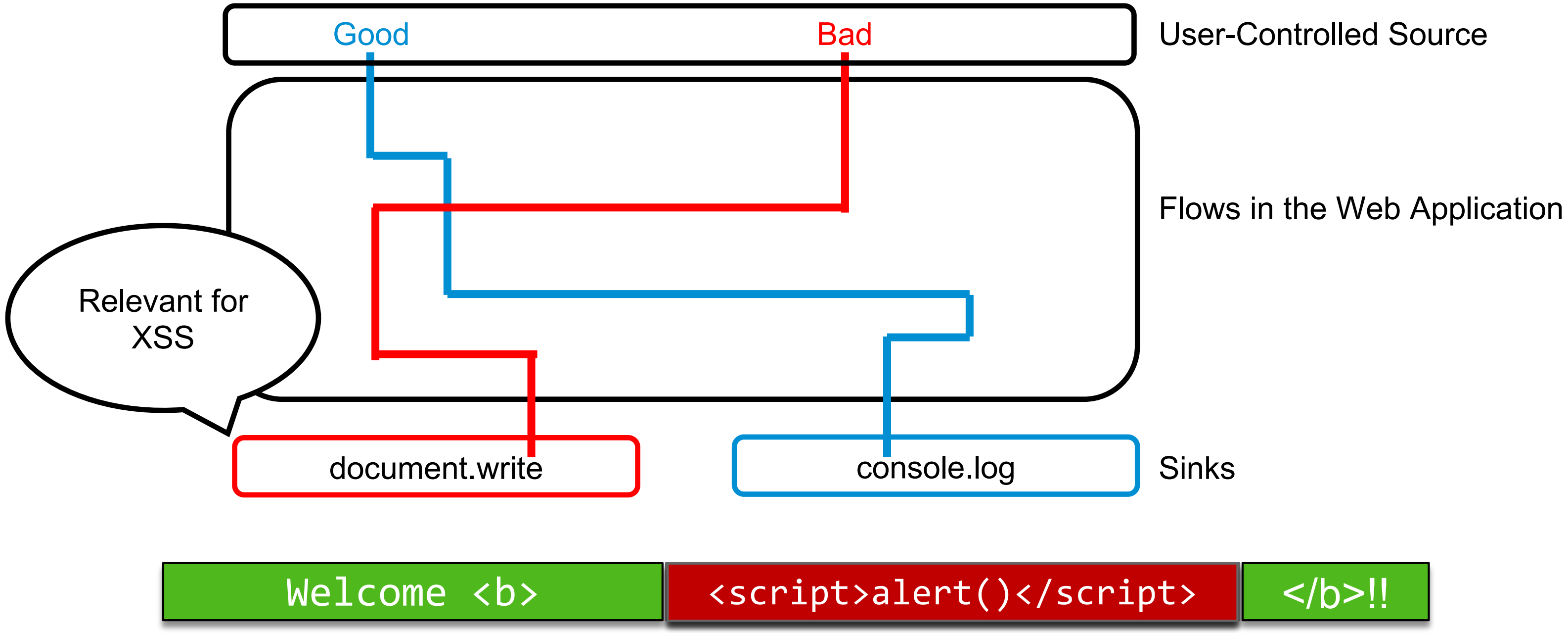
1: Motivation

OWASP Top Ten Web Application Security Risks

OWASP Top 10 - 2017
A1:2017-Injection
A2:2017-Broken Authentication
A3:2017-Sensitive Data Exposure
A4:2017-XML External Entities (XXE)
A5:2017-Broken Access Control
A6:2017-Security Misconfiguration
A7:2017-Cross-Site Scripting (XSS)
A8:2017-Insecure Deserialization
A9:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring

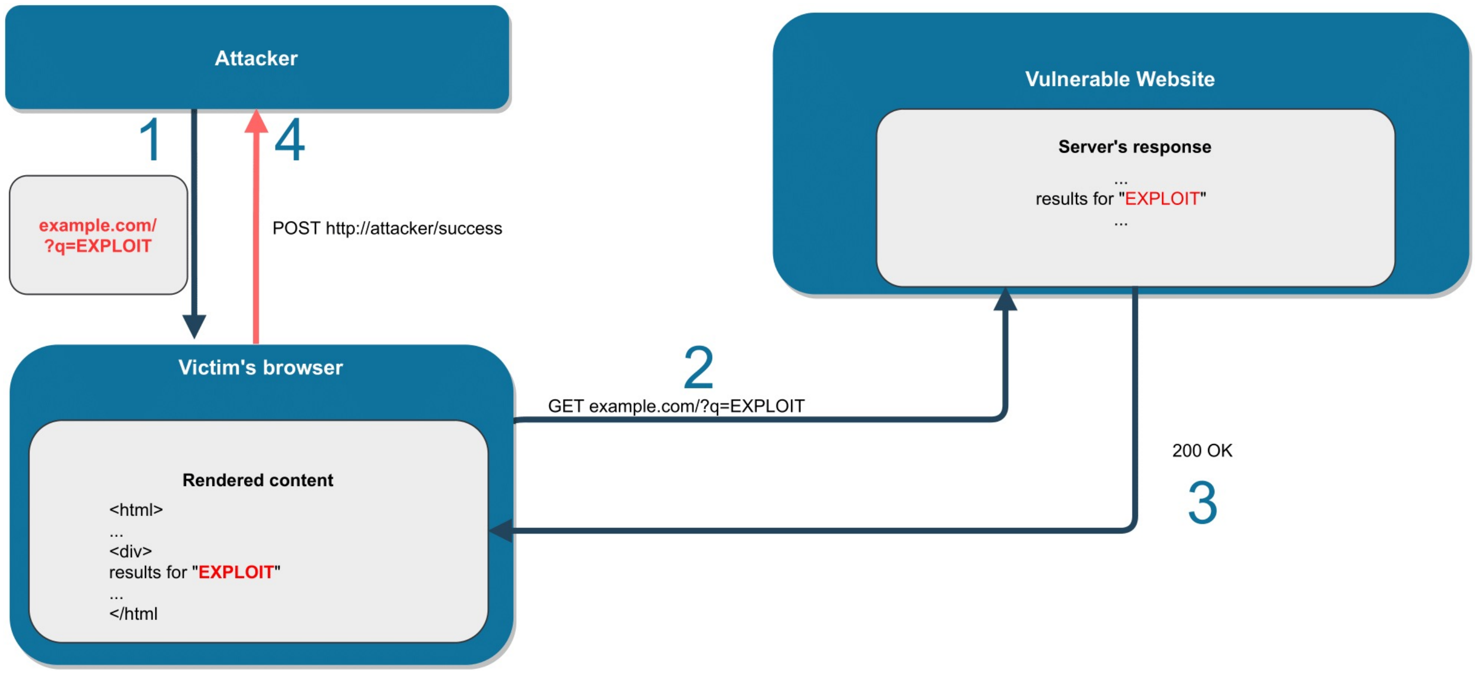
Example of a Vulnerable Flow

```
document.write("Welcome <b>" + location.hash.substr(1) + "</b>!!");
```



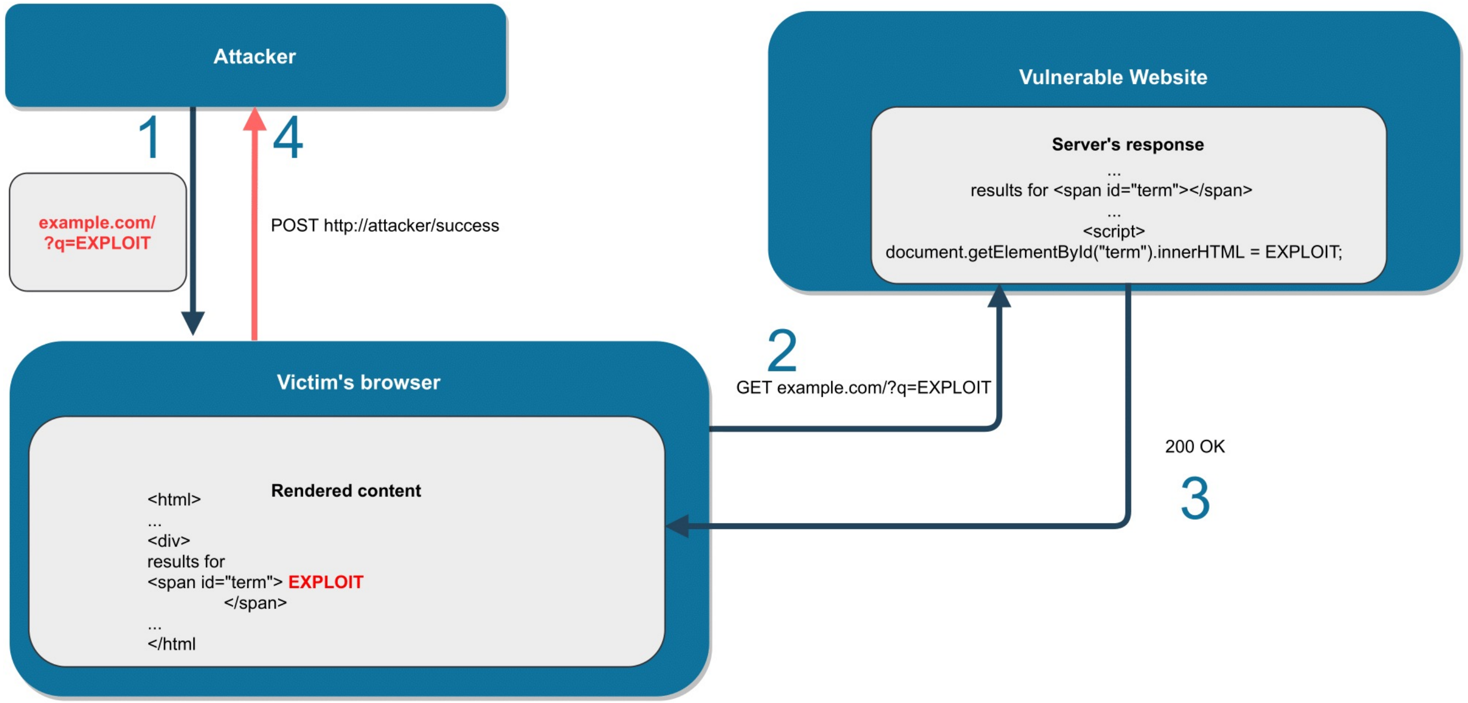
Reflected Cross-Site Scripting

- The vulnerable Web Application echoes a user-provided input without a proper sanitization. The malicious script is inserted into the page in the server side.



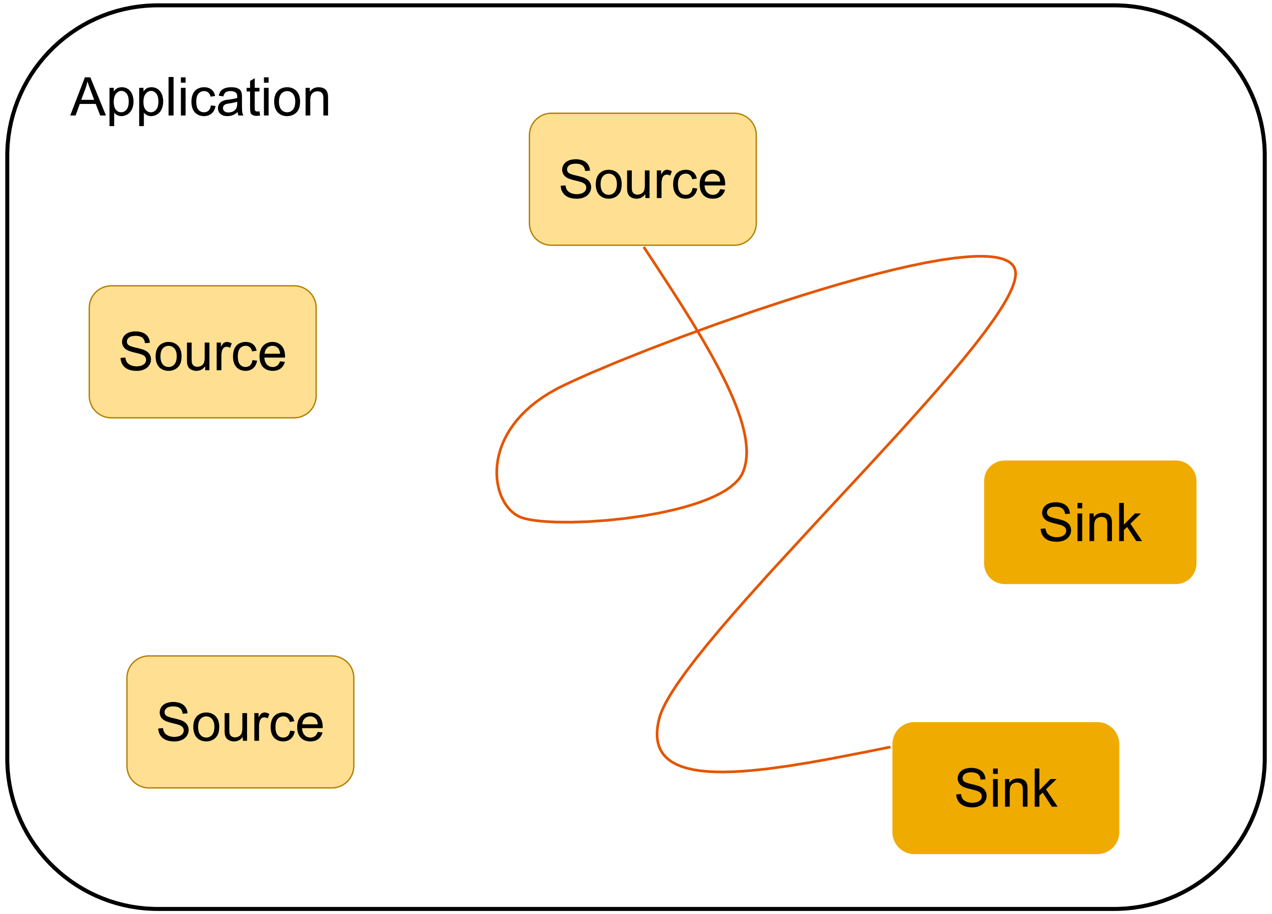
Client-Side Cross-Site Scripting

- The vulnerable Web Application echoes a user-provided input without a proper sanitization. The malicious script is inserted into the page in the client side.



2: Tainting in the Client Side

The Taint-Aware Browser



- Firefox Browser enhanced with taint analysis capabilities.
  - Enhanced JavaScript engine to propagate taint information for Strings.
  - String is tainted if it originates from a source function.
  - If a tainted string enters a sink function, it triggers a JS event.
- Modified browser that supports dynamic, byte-level taint-tracking of suspicious flow.
- Information about encoding and decoding functions is stored.

Problematic

- The payload <script>evilFunction()</script> doesn't work for all cases.

```
document.write("<img src='//adve.rt/ise?hash=" + location.hash.slice(1)+ "'/>");
```

```
<img src='//adve.rt/ise?hash= '/> <script>alert(1)</script> '/>
```

Idea

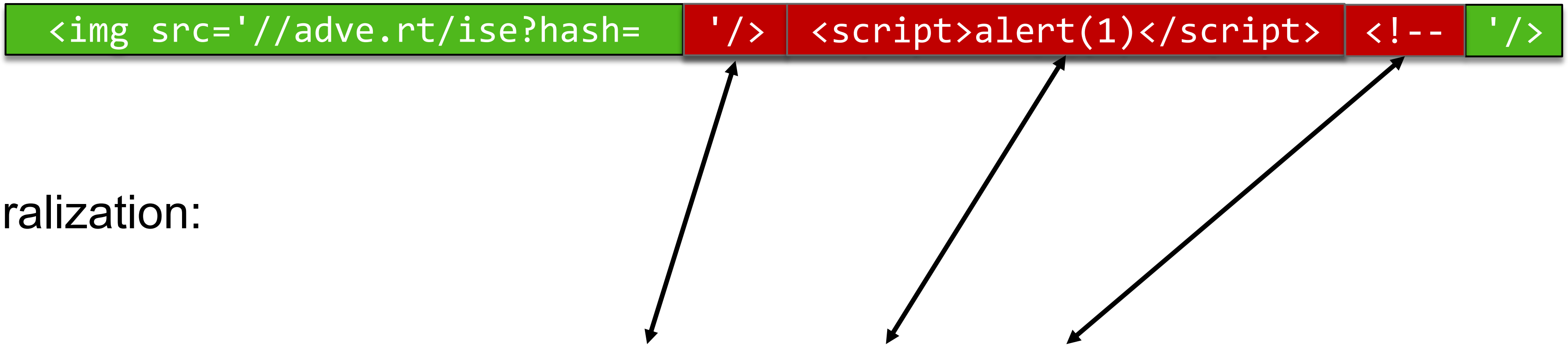
- Automated generation of exploits using the tainting analysis done with the developed taint-aware browser.
- Improvement of the solutions already existent in the literature.

3: Automated Generation of Cross-site Scripting Exploits

Structure of an Exploit

- Example:

```
document.write("<img src='//adve.rt/ise?hash=" + location.hash.slice(1)+ "'/>");
```

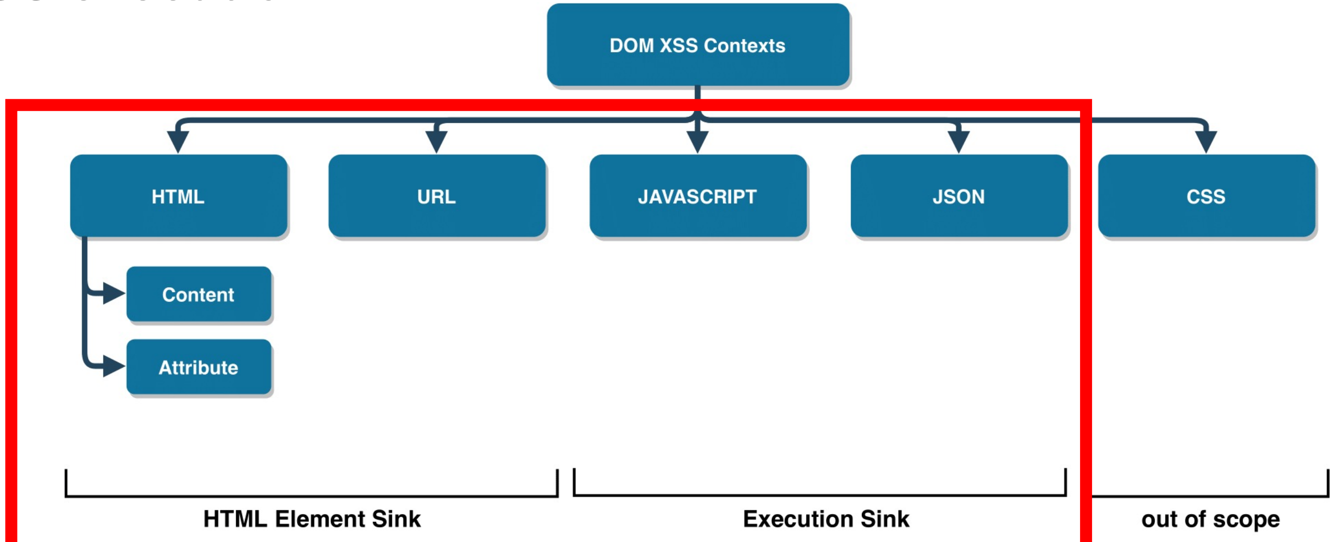


- Generalization:

$$exploit \coloneqq breakOut + payload + breakIn$$

Context Dependent Generation

Contexts for XSS execution



document.write
document.writeln
innerHTML
insertAdjacentHTML
outerHTML

eval
script.text
setTimeout
setInterval

Paper Scope

Context Dependent Generation

The HTML Element Sink : Break-Out Sequence

Form the break-Out sequence

$$breakOut \coloneqq contextPrefix + closingCurrentTag + closingTags$$

- contextPrefix: add a # character or not.
- closingCurrentTag: using an HTML parsing tree.
- closingTags: </iframe></style></script></object></embed></textarea>

Context Dependent Generation

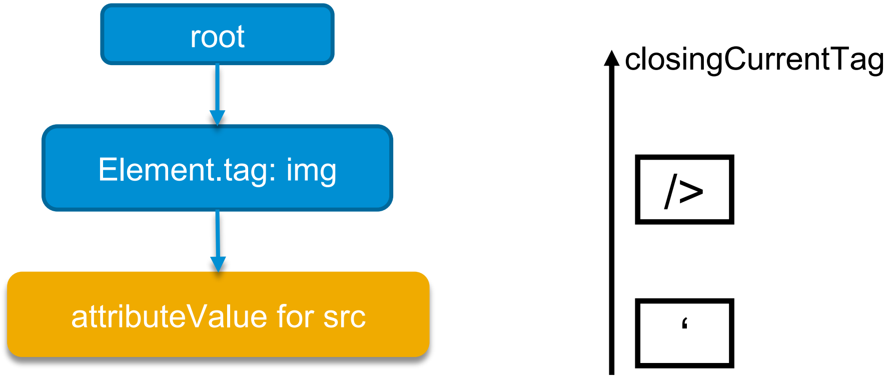
The HTML Element Sink : Break-Out Sequence

Example for constructing the closing tag

$$breakOut \coloneqq contextPrefix + closingCurrentTag + closingTags$$

```
document.write("<img src='//adve.rt/ise?hash=" + location.hash.slice(1)+ "'/>");
```

```
<img src='//adve.rt/ise?hash= | 123 | '/>
```



$exploit \coloneqq breakOut + payload + breakIn$

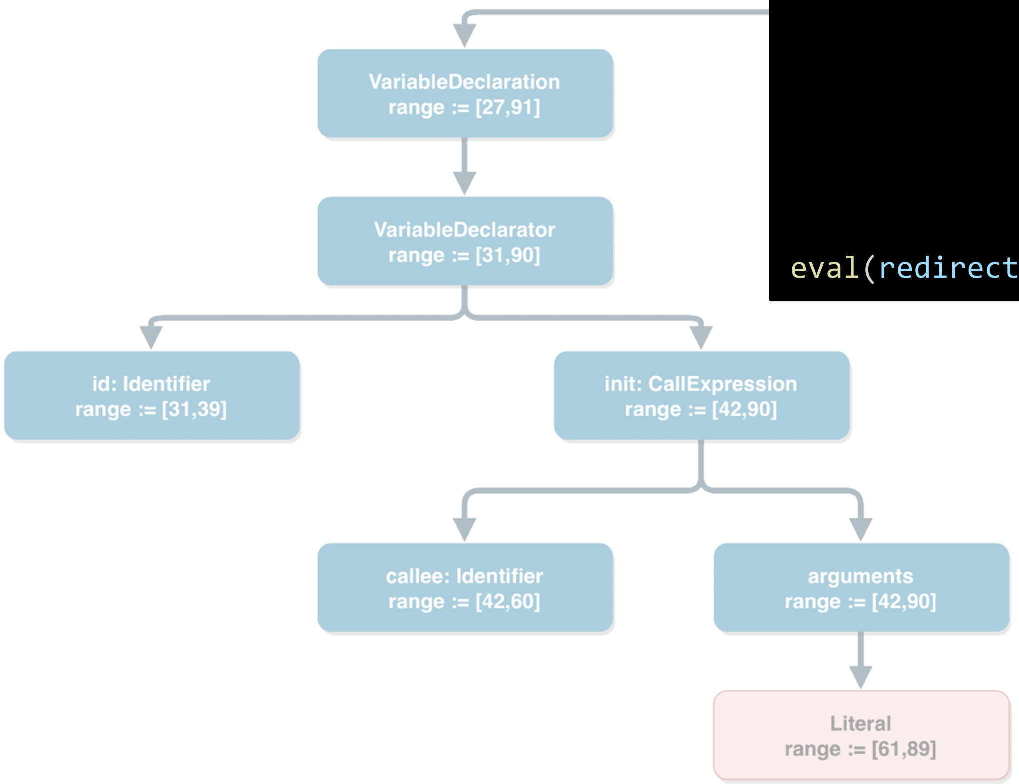
Context Dependent Generation

The Execution Sink : Break-Out Sequence

Step 1: JS parsing tree.

```
let redirection = 'function redirect() {'
                + 'let redirURL = decodeURIComponent(' + window.location.hash + ');'
                // inside the block statement
                + 'if (redirURL) {'
                + 'window.location = redirURL;'
                + '}'
                + '}';
                // top level

eval(redirection);
```



Context Dependent Generation

The Execution Sink : Break-Out Sequence

Step 2: Form the Break-Out Sequence

$$breakOut \coloneqq contextPrefix + closingCurrentNodes$$

- contextPrefix: add a # character or not
- closingCurrentNodes

Context Dependent Generation

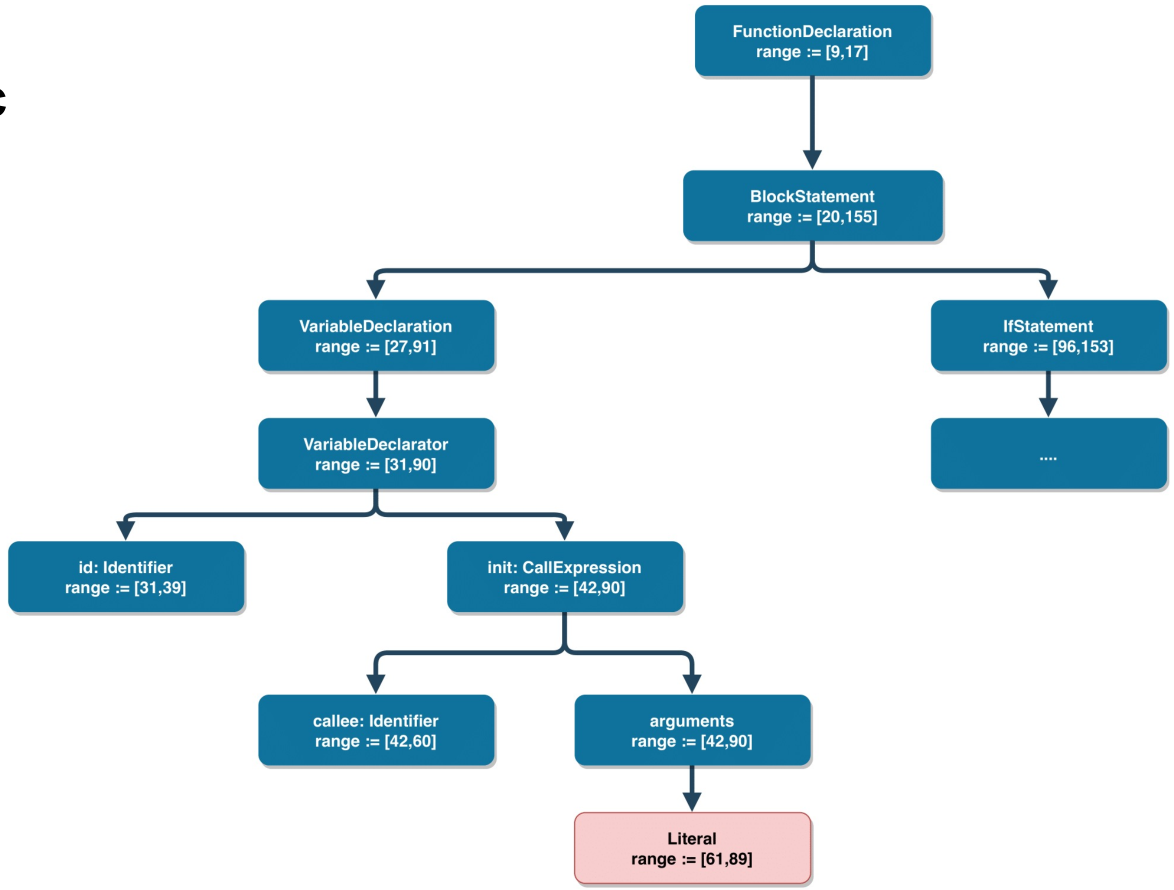
The HTML Element Sink : Break-Out Sequenc

Example for constructing the closing nodes

$$breakOut \coloneqq contextPrefix + closingCurrentNodes$$

1. Literal: ‘
2. Arguments:
3. CallExpression:)
4. VariableDeclarator:
5. VariableDeclaration: ;
6. BlockStatement: }
7. FunctionDeclaration: ;

```
let redirection = 'function redirect() {'
                + 'let redirURL = decodeURIComponent('');};payload;
```



Context Dependent Generation

The Execution Sink : Break-Out Sequence

Step 3: Syntax check for the generated breakOut

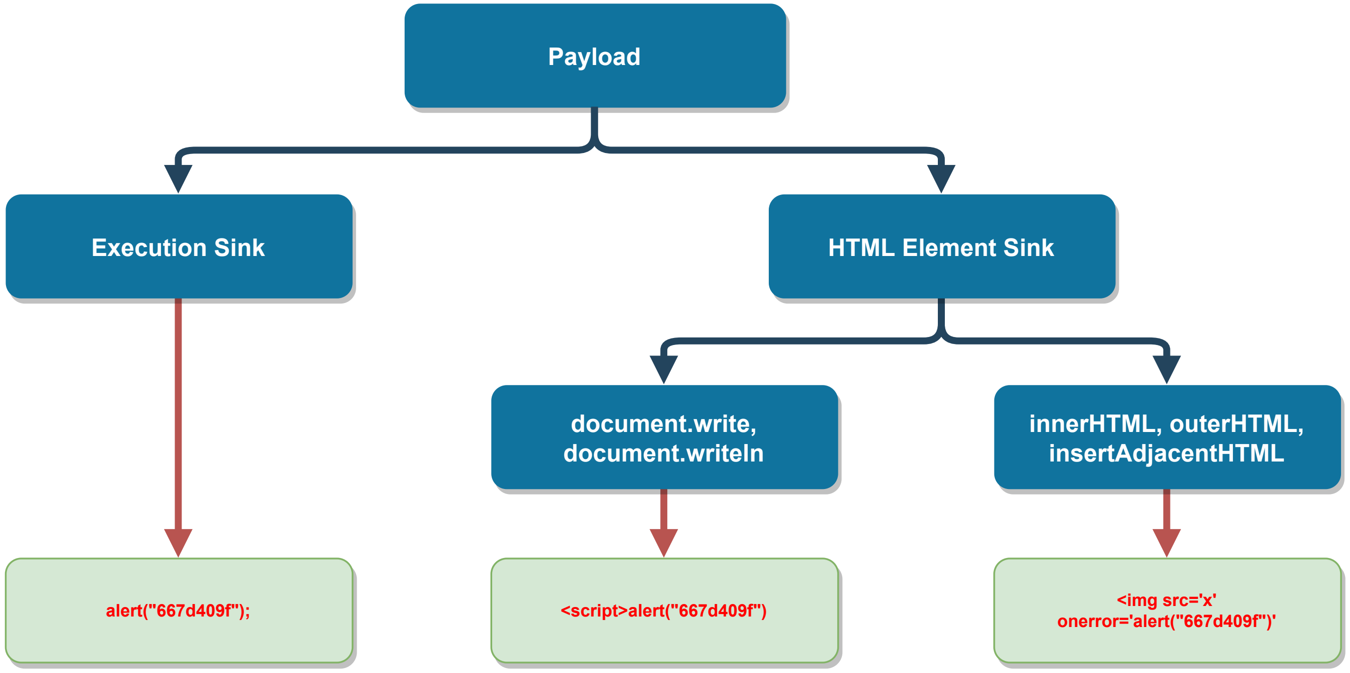
```
let redirection = 'function redirect() {'
                + 'let redirURL = decodeURIComponent('');};
```

$$exploit \coloneqq breakOut + payload + breakIn$$

Context Dependent Generation

The payload

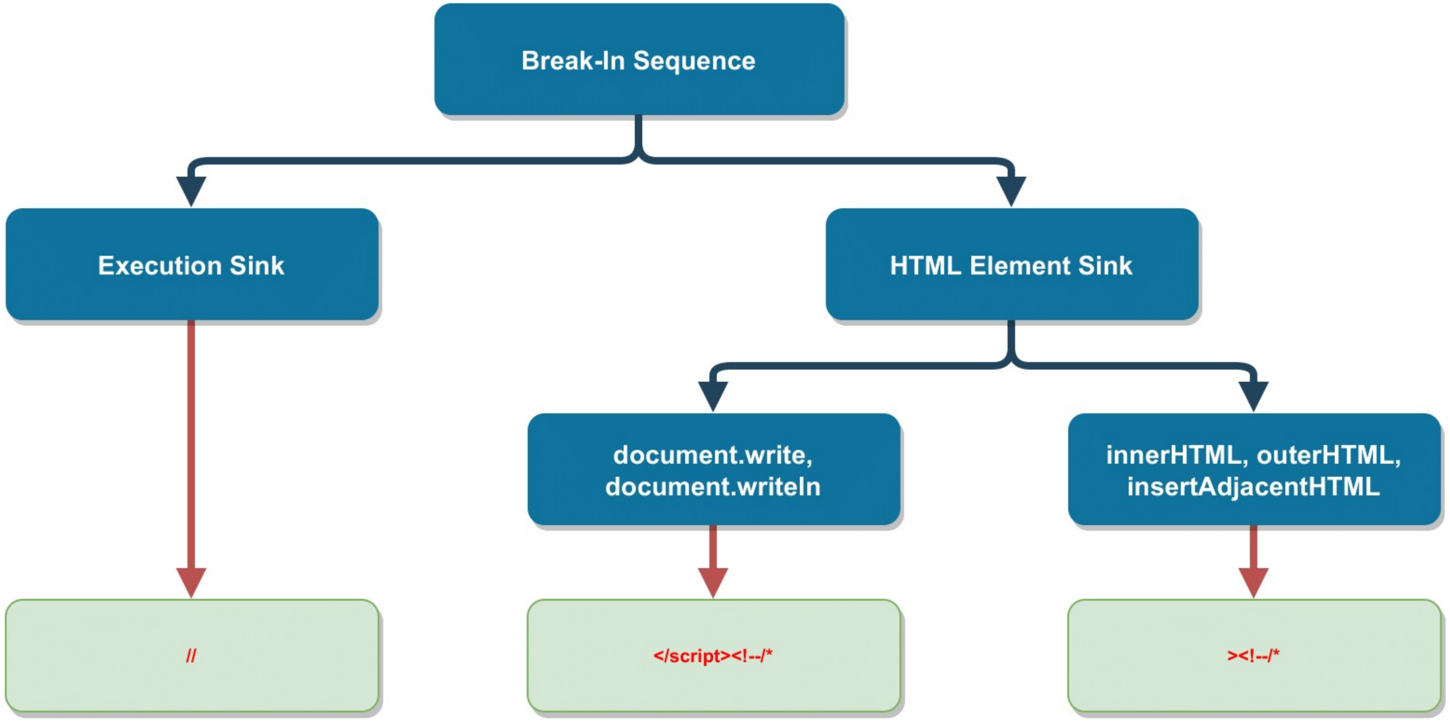
$$payload \coloneqq enterScriptContext + alert(id)$$



$$exploit \coloneqq breakOut + payload + breakIn$$

Context Dependent Generation

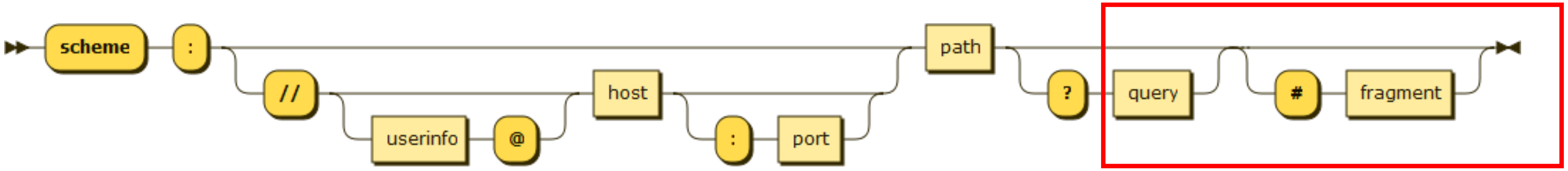
The Execution Sink : Break-In Sequence



Context Dependent Generation

The Injection Mechanisms

- The exploit is now generated
- Question: Where to inject this exploit exactly in order to increase the success validation rate.
- URL Structure[1]:



- 2 previous methods from the literature are re-implemented and compared with our method.

1 https://en.wikipedia.org/wiki/URL

Context Dependent Generation

http://example.com/1?payload=**abcd**&sp=x

The Injection Mechanisms: Method A [1]

- Appends the URL with a fragment containing the generated exploit

Method A: http://example.com/1?payload=**abcd**&sp=x#**EXPLOIT**

1 Sebastian Lekies, Ben Stock, and Martin Johns. "25 million flows later: Large-scale detection of DOM-based XSS". In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013, pp. 1193–1204.

Context Dependent Generation

http://example.com/1?payload=**abcd**&sp=x

The Injection Mechanisms: Method B [1]

- Moves the query string in question to the fragment and changes its value to an exploit.

Method B: http://example.com/1?sp=x#&payload=EXPLOIT

1 William Melicher et al. “Riding out domsday: Towards detecting and preventing dom cross-site scripting”. In: 2018 Network and Distributed System Security Symposium (NDSS). 2018.

Context Dependent Generation

http://example.com/1?payload=**abcd**&sp=x

The Injection Mechanisms: Method C

- Makes use of the information about the tainted flow to resolve in which context it exits within the URL and then calculates the replace ranges.

Method C: http://example.com/1?payload=EXPLOIT&sp=x

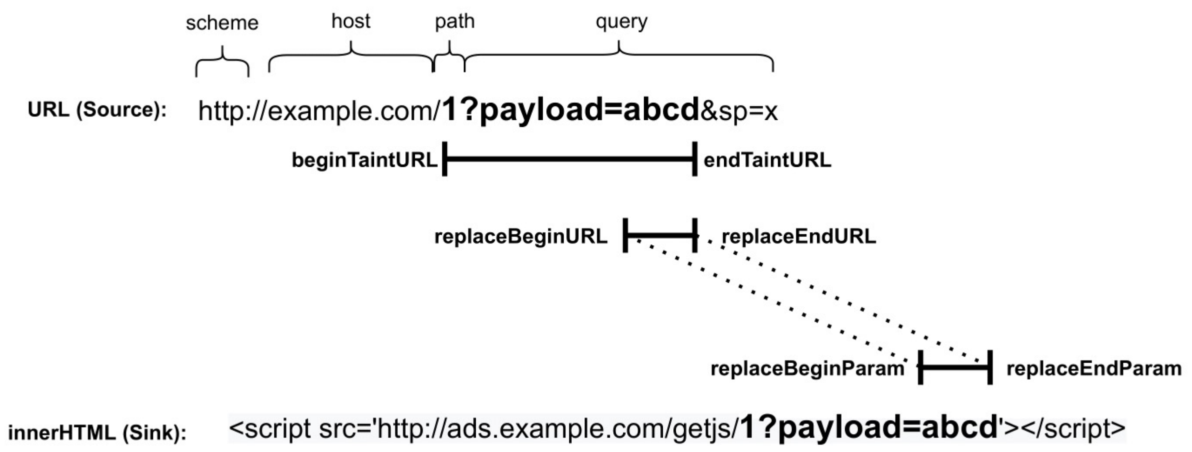
- Hypothesis: More precise than the two other methods.
- Doesn’t always make use of the fragment to pass the exploit.

Context Dependent Generation

The Injection Mechanisms: Method C in depth

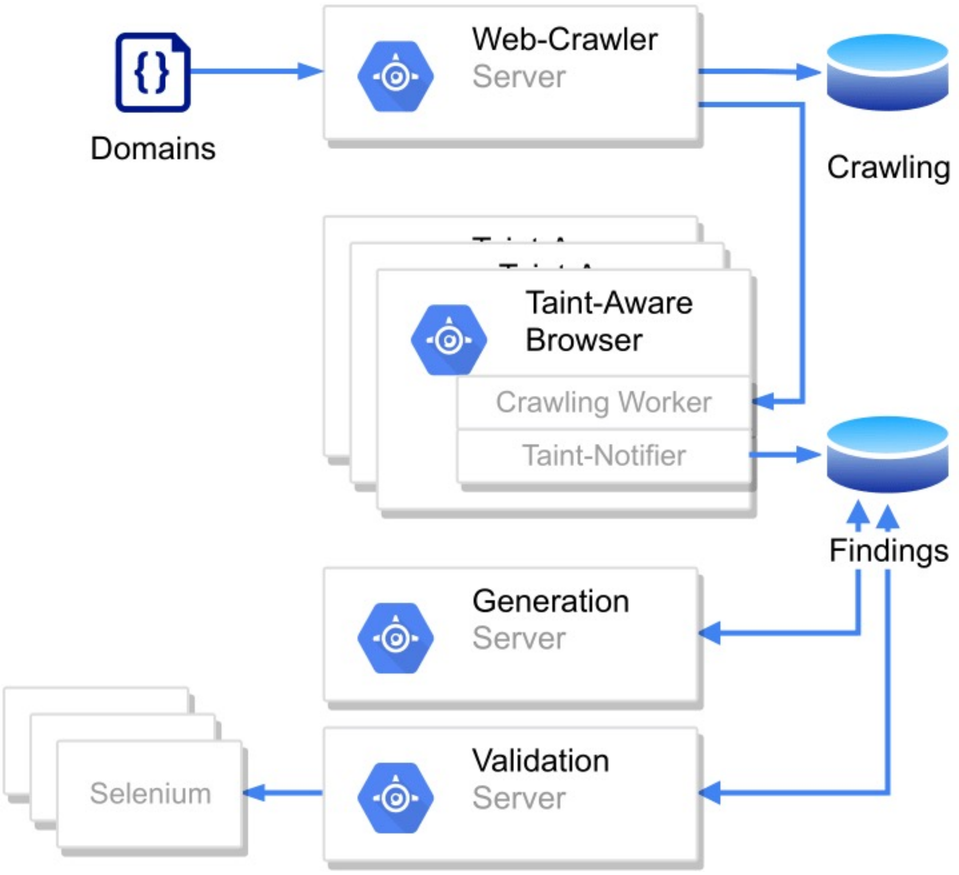
- This method consists of calculating the following indices:

  - **beginTaintURL and endTaintURL**
  - **replaceBeginURL and replaceEndURL**
  - **replaceBeginParam and replaceEndParam**



4: Large Scale Crawling and Results

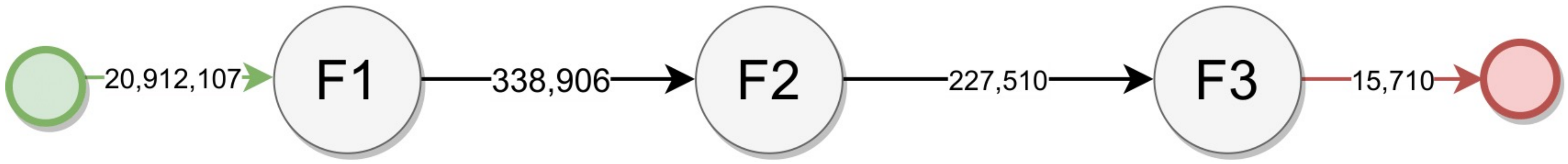
Big Picture and Orchestration



The Crawling Results

	This work (Method C)	Melicher et al. (Method B)	Lekies et al. (Method A)
Date	2020	2017	2013
Seed domains	100,000	10,000	5,000
Subpages up to	10	5	All sub pages from depth 1
Web pages	390,092	44,722	504,275
Pages / domain	3.90	4.47	100.86
Frames	1,111,821	319,481	4,358,031
Taint Flows	20,912,107	4,140,873	24,474,873
Flows / Page	**53.61**	**92.59**	**48.53**

Pre-Generation Filters



F1: preserve only flows with URL-based sources and HTML/JS sinks.

F2: exclude flows containing escape, encodeURI or encodeURIComponent.

F3: remove duplicated flows.

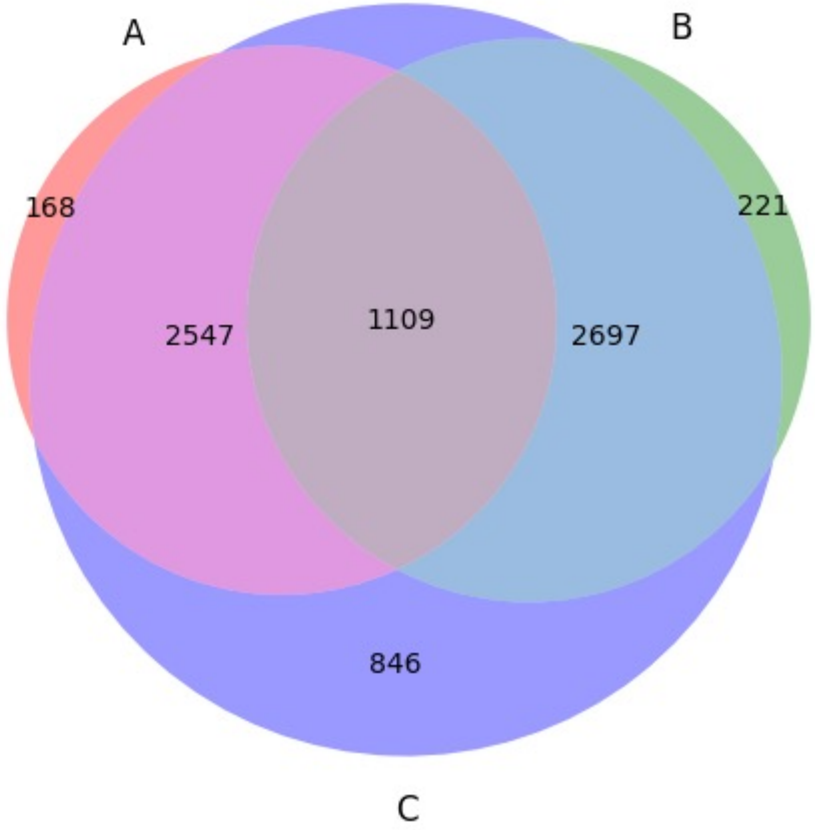
Generation and Validation Results* Comparison

	Method C	Method B	Method A
Flows for which an exploit was generated	15,710	5786	15,710
Flows for which an exploit was validated	7199	4041	3838
Flows successfully validated / Flows relevant for generation	**45.82%**	**25.72%**	**24.43%**
URLs generated	126,332	57,393	15,710
URLs validated	16,993	8628	3838
URLs successfully validated / URLs generated	13.45%	15.03%	24.43%

* All methods evaluated using our dataset

Venn Diagram of the Validated Flows

- **94.87%** of the validated flows could be validated with Method C.
- 11.15% could be validated only with method C.
- Total success rate of the three methods combined: 48.3%
- 8122 Findings couldn't be validated by any of the three methods.



5: Conclusion

Conclusion

- Client-side XSS is gaining more relevance as the frontend technologies are developing fast.
- A novel technique for exploiting client-side XSS is presented.
- From 15,710 findings relevant for a successful exploit, 7199 (48.3%) could be validated.
- Improvement over the two previous exploit generation techniques by factors of 1.9 and 1.8.

Thank you.
Any Questions?

Contact information:

Souphiane Bensalim
SAP Security Research
souphianebensalim@gmail.com

Thomas Barber
SAP Security Research
thomas.barber@sap.com

David Klein
Technische Universität Braunschweig
david.klein@tu-braunschweig.de

Martin Johns
Technische Universität Braunschweig
m.johns@tu-braunschweig.de